Modular Programming Techniques for Distributed Computing Tasks

Anthony Cowley, Hwa-Chow Hsu, Camillo J. Taylor
GRASP Laboratory
University of Pennsylvania, Philadelphia, PA, USA, 19104

ABSTRACT

This paper describes design patterns used in developing a software platform for mobile robot teams engaged in distributed sensing and exploration tasks. The goal of the system presented is to minimize redundancy throughout the development and execution pipelines by exploring the application of a strong type system to both the collaborative development process and runtime behaviors of mobile sensor platforms. The solution we have implemented addresses both sides of this equation simultaneously by providing a system for self-describing inputs and outputs that facilitates code reuse among human developers and autonomous agents. This well-defined modularity allows us to treat executable code libraries as atomic elements that can be automatically shared across the network. In this fashion, we improve the performance of our development team by addressing software framework usability and the performance and capabilities of sensor networks engaged in distributed data processing. This framework adds robust design templates and greater communication flexibility onto a component system similar to TinyOS and NesC while avoiding the development effort and overhead required to field a full-fledged web services or Jini-based infrastructure. The software platform described herein has been used to field collaborative teams of UGVs and UAVs in exploration and monitoring scenarios.

KEYWORDS: *sensor network, distributed computing, software design*

1. INTRODUCTION

As efforts to field sensor networks, or teams of mobile robots, become more ambitious [5], [11], [4], communication constraints rapidly become the bottleneck both in the development effort and execution environment. From a development standpoint, human networking becomes clumsy as team sizes grow, putting team communications at a premium. Therefore, effort should be spent to optimize away the time developers must spend explaining things to each other, specifically, how to write code that has already been written or how to reuse existing code. If this aspect of collaborative development is not explicitly addressed, the team runs the risk of either losing the ability to reuse code, due to a lack of shared understanding, or drastically curtailing productivity by devoting excessive time to documentation efforts. Ideally, each developer's efforts will be documented extensively enough for others to easily reuse the existing code without placing an undesirable documentation burden on the original developer.

The desire for software agents to autonomously exploit existing code is a subtly parallel goal. Should an agent be able to specify its requirements, it ought to be able to identify any existing code that would meet this need. This applies both in the sense of agents discovering new sources of data, and that of interactive data processing requests. We wish to field a sensor network wherein one sensor can tap into a potentially live data stream without any a priori knowledge of other nodes or their capabilities, while also giving each node on the network the ability to ask questions that require the processing of large amounts of data. In the first case, we need to give our agents the ability to identify the types of data being exported by other agents. This is addressed by having communication endpoints describe the data they trade in. The latter case involves not only finding the correct type of data, but also sending an active query to the data rather than saturating the network by bringing the data to the query. Such behavior requires descriptions of data sources and sinks, as well as the ability to move, command, and control executable code across the network.

2. IMPLEMENTATION

A crucial aspect in the development of this framework design philosophy is the relationship between the new software and that which it is built upon. We chose to develop our high level environment on top of an already full-featured platform. In our case, this platform was Microsoft's .NET technology, which includes a strong type system in the .NET CLR (Common Language Runtime), an object-oriented language in the form of

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
AUG 2004		**00-00-2004 to 00-00-2004**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Modular Programming Techniques for Distributed Computing Tasks	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Pennsylvania,GRASP Laboratory,Philadelphia,PA,19104**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
Proceedings of the 2004 Performance Metrics for Intelligent Systems Workshop (PerMIS ?04), Gaithersburg, MD on August 24-26 2004

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**8**	

C#, and many varieties of network functionality in the .NET Class Library. Our design then focused both on what functionality we wished to add and that which we wished to remove. Simply put, we want to impose some structure on our developers that is not inherent to C#, .NET, or any existing platform. This structure is a fundamental part of the ROCI (Remote Objects Control Interface) [6], [9] philosophy, and is imposed on the ROCI developer as a form of design control that we believe adds a level of reliability to the resultant system. By imposing a prescribed design on developers, we are better able to isolate potential weaknesses and build in error detection and handling functionality.

ROCI itself is a high level operating system useful for programming and managing sensor networks. The core control element in the ROCI architecture is the ROCI kernel. A copy of the kernel runs on every entity that is part of the ROCI network (robots, remote sensors, etc.). The kernel is responsible for handling program allocation and injection. It allows applications to be specified and executed dynamically by forming communication connections and transferring code libraries to the nodes as needed. The kernel is also responsible for managing the network and maintaining an updated database of other nodes in the ROCI network. In this way, ROCI acts as a distributed peer-to-peer system. Nodes can be dynamically added and removed from the network, and information about these nodes and the code running on them is automatically propagated throughout the system without the need for a central repository.

The control functionality needed by such a kernel is made possible by self-contained, reusable modules. Each module encapsulates a process which acts on data available on its inputs and presents its results on well defined outputs. Thus, complex tasks can be built by connecting inputs and outputs of specific modules. These connections are made through a pin architecture that provides a strongly typed, network transparent communication framework. A good analogy is to view each of these modules as an integrated circuit (IC) that has inputs and outputs and does some processing. Complex circuits can be built by wiring several ICs together, and individual ICs can be reused in different circuits. ROCI modules have been developed for a wide range of tasks such as: interfacing to low level devices like GPS units and cameras, computing position estimates based on GPS, IMU and odometry data, acquiring stereo panoramas, platform motion control, online map building and GPS waypoint navigation.

ROCI modules are further organized into tasks (Figure 1). A ROCI task is a way of describing an instance of a collection of ROCI modules to be run on a single node, and how they interact at runtime. Tasks represent a family of modules that work together to accomplish some end goal – a chain of building blocks that transforms input data through intermediate forms and into a useful output. A task can be defined in an XML file which specifies the modules that are needed to achieve the goal, any necessary module-specific parameters, and the connectivity between these modules. Tasks can also be defined and changed dynamically by starting new modules and connecting them with the inputs and outputs of other modules.

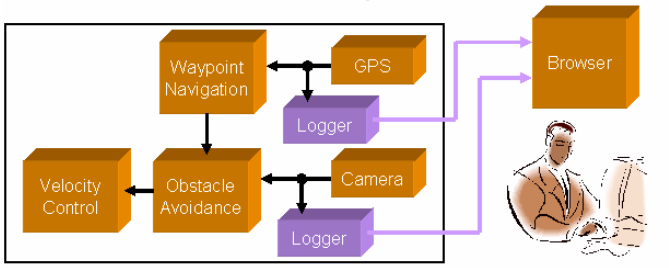

Figure 1. A typical ROCI task: a collection of behavior modules with loggers connected to specific pin connections. A human operator interfaces with the logs via the browser, which may be running on a different machine from the task.

The wiring that connects ROCI modules is the pin communication architecture. Pin communications in ROCI are designed to be network transparent yet high performance. Basically, a pin provides the developer with an abstract communications endpoint. These endpoints can either represent a data producer or a data consumer. Pins in the system are nothing more than strongly typed fields of the module class, and so are added to modules with a standard variable declaration statement. Pin communication allows modules to communicate with each other within a task, within a node or over a network seamlessly. The base Pin type will optimize the connection based on whether or not it is local and handle all error detection and handling, bandwidth utilization requirements, and optional buffering. The type system enforces pin compatibility at

run time which makes it impossible to connect inputs and outputs of incompatible types.

This compatibility evaluation is done in an object-oriented fashion such that, when necessary, output data will be transparently up-cast before being transmitted to a data sink. This negotiated compatibility allows for what we call "blind polymorphism," which does not require that both nodes have all the same types loaded. That is to say, if data can be cast up its inheritance hierarchy to the type that the data sink requires, then this cast will be done on the source side of the connection, thereby not requiring that the sink be aware of the inherited type.

Importantly, the modules in the system are self describing so that the kernel can automatically discover their input and output pins along with any user-settable parameters. These features of the ROCI architecture facilitate automatic service discovery since a module running on one ROCI node can query the kernel database to find out about services offered by modules on other nodes and can connect to these services dynamically.

The self describing behavior of module inputs, outputs, and parameters is achieved automatically through the use of the underlying type system. This is an important element of ROCI's ability to limit the potential for developer error. In the process of identifying necessary input and output pins, the module developer naturally defines certain data structures that the module takes as input and generates as output. These data structures represent a form of design contract that tells other users what type of input the module can parse, and what type of output it generates. This information is what the pin type system is built upon: a particular type of pin is designed to transfer a particular type of data. These types can then be used to verify potential connections between pins. By relying on type information that the developer necessarily creates by designing module-appropriate data structures, we are able to obviate the need for any separate developer-generated description of a module's inputs and outputs. Such descriptions run the risk of becoming out of date, and are not always easily checked. Relying on the type system, however, means that if a module incorrectly parses an input data structure, for example, it will not compile. In this way we guarantee that if a Module compiles, then it must be compatible with the associated data description.

2.1. *The Task Programming Model*

The abstraction gained by treating modules as primitive components allows us to bring compiler-level features to bear on ROCI tasks. Specifically, the idea of type checking the input/output connections between modules has already been covered, but type checking the parameters that govern the behavior of these modules is also provided at the task level.

Individual module authors are able to decorate class-scope variable declaration statements with attributes that specify whether or not a variable is a startup parameter, or even if it is a control parameter that should be modifiable at run time. These attributes are extracted from compiled code, and are used by the ROCI to kernel to expose these variables when appropriate.

Variables marked as startup parameters will be displayed in the browser UI when a user wishes to start a task. Type checking is performed as the user enters new values for these parameters, thus making it far less likely that a module will start with invalid parameters. Furthermore, the type of the parameter can be used to intelligently populate the parameter-setting UI by dynamically creating UI elements such as drop-down boxes with only valid values as options, as opposed to a text field for every parameter. Variables marked as control parameters (dynamic over the course of execution) can be modified by another standard browser interface. A running module can be selected, and any variables marked as control parameters will populate a parameter-setting UI similar to the one described for startup parameters. This functionality, built atop the strong type system in .NET, provides a compiler-like layer of type checking at all phases of execution, while simultaneously making the UI used to interact with a ROCI deployment more intuitive for the end user.

2.2. *General Instrumentation*

The notion of task as program allows for varied interesting forms of system-level instrumentation and control. First, by sufficiently isolating individual modules such that they can be treated as atomic operations, we are able to treat tasks as programs built on a language that uses the specified modules as statements. Second, by virtue of its role as provider of

all inter-module communications, the ROCI kernel is capable of rich monitoring and control of all data transactions. These two points both deal with the notion of program flow control.

Program flow control is primarily controlled by the sequence of operations specified in the program. In our case, a schedule of modules makes up the procedural part of a task program. As described above, a task is a collection of concurrently running modules. The order in which these modules run is not explicitly defined, but instead is effectively governed by data dependencies between modules. In general, if module alpha uses data from module beta, then module alpha will block until that data is available, thus creating a very loose schedule in which each iteration of module alpha's processing loop is preceded by at least one iteration of module beta. There are no guarantees on the efficiency of this schedule; if only module alpha uses module beta's output, then it may be wasteful for module beta to run at a higher rate than alpha.

This issue is addressed by having a task schedule. The task schedule merely specifies a linear sequence of module iterations, but can be leveraged to obtain far greater efficiency that a schedule governed solely by dependency blocking. This schedule is specified in the task XML file as a sequence of module names. The names are checked when the task file is loaded to ensure that all statements in the schedule are defined module names. This schedule can be used simply to eliminate wasted iterations of data producers, but it can also be used to obtain non-obvious gains in overall program efficiency. A schedule can include a bias to run a particular module more frequently than another if it would give the task, taken as a whole, greater efficiency. Furthermore, since this schedule is not encoded in compiled code, it is fully dynamic. That is, a user or automated process can adjust a task's schedule at runtime to meet changing resource availability or execution priorities.

Such behavior is dependent on information. This information is made available by the instrumentation built into task schedules. The mechanisms that govern the execution of a ROCI task are in good position to monitor the iteration frequency of the task schedule in its entirety, and the resources being used by individual modules. This information can be used to raise alarms when a task frequency drops below a specified threshold, to throttle iteration frequency, or to modify the schedule to make better use of available resources. Furthermore, application specific efficacy metrics can be utilized by task monitoring modules to initiate new schedules to improve efficiency.

The distributed nature of ROCI deployments suggests a form of program flow throttling apart from the usual method of CPU resource allocation: network resource allocation. While the scheduling system can be used to monitor and control the rate at which a task schedule iterates, ROCI's pin system can throttle network communications on a connection-by-connection basis. Individual pin connections can be monitored to examine the type of data being transmitted, the frequency of transmissions, and the bandwidth used. Both the frequency of transmission and the overall bandwidth used are controllable by the ROCI kernel. This allows a controller, human or automated, to give network precedence to certain connections, potentially allowing greater system effectiveness with limited resources. Note that by throttling network communications, the speed at which a networked task runs can be controlled. Especially in a schedule-free execution environment, wherein a collection of modules have their iteration frequencies mediated by data dependencies, the throttling of individual connection bandwidth can be used to control the iteration frequency of individual modules. Thus there are two distinct methods of controlling performance in an on-demand fashion based on mediating CPU or network resource allocation.

2.3. *Logger Modules*

Our sensor database [12], [10] system is implemented on top of ROCI through the addition of logger modules. These logger modules can be attached to any output pin and record the outputs of that pin's owner module in a time-stamped log which can be accessed by external processes. These logger modules appear to the system as regular ROCI modules which means that they can be started and stopped dynamically and can be discovered by other ROCI nodes on the system. This last point is particularly salient since it means that robots can learn about the records available in other parts of the network at run time as those resources become available. Since logger modules can be attached to any output pin, there is no meaningful distinction between "low level" sensor data such as images returned by a camera module and "high level" information such as the output of a position

estimation module. Any data that is relevant to a task can easily be logged through the addition of a logger module.

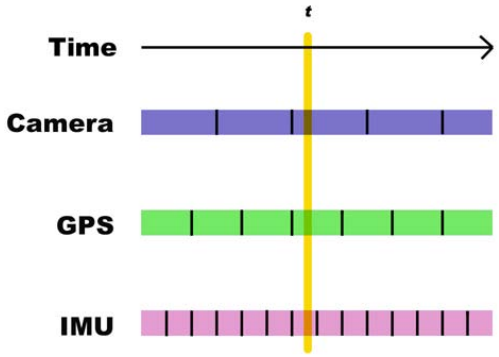

Figure 2. Time is a useful index for synchronizing data concurrently collected from multiple sources.

The generic logger module logs all incoming data based on time, an index relevant and meaningful regardless of the data type (Figure 2). Additional indexing methods that are specific to a particular data type are easily implemented by creating a new type of logger module that inherits from the general logger and is explicitly usable only with the expected data type. For example, a logger module that records the output of a GPS unit may also support efficient indexing based on position. Using time as a common key provides a simple mechanism for correlating information from different channels. Consider, for example, the problem of obtaining all of the images that a robot acquired from a particular position. This can be implemented efficiently by first indexing into the GPS log to find the times at which the robot was at that location and then using those times to index the image log to pull out the images taken from that vantage point. Using time as a common index also eliminates the need for a fixed database schema on the robots: different logger modules can be added or removed from a node as needed without having to perform complex surgery on a global table of sensor readings. Since the logger processes do not interact directly, they can be started and stopped, added and removed independently of each other.

2.4. *Query Processing*

Once a relevant data log has been found on the network, one must then face the problem of executing a query to extract information from that archive. It is often the case that the volume of data stored in a log makes it unattractive to transfer the data over the network for processing. In these situations we can take advantage of the fact that the facilities provided by ROCI can be used to support distributed query processing. Consider the example of a UAV that stores a log of images acquired as it flies over a site. If a process on a UGV wanted to access this data to search for particular targets in the scene, it would be impractical to transfer every image frame to the ground unit for processing. Here it makes sense to consider sending an active query to the UAV requesting it to process the images and send the target locations back to the UGV. This can be accomplished by developing a ROCI module that extracts the targets of interest from UAV imagery and then sending this module to the UAV as part of a query. The ROCI kernel on the UAV would then instantiate a task and use this module to process the data in the image log returning the results to the UGV.

Sophisticated queries that involve chaining together the results of many processing operations or combining information from several logs can be handled through precisely the same mechanism. The query takes the form of a network of ROCI modules that carry out various phases of the query. The modules in this task are distributed to appropriate nodes on the network and the final output is returned to the node that initiated the request. This approach allows us to dynamically distribute the computation throughout the network in order to make more efficient use of the limited communication bandwidth.

Another feature of this approach is that it promotes code re-use since the modules that are developed for carrying out various data processing and analysis operations online can also be used to implement queries on stored data logs (Figure 3). This is important not just by virtue of facilitating rapid development, but also by the robustness and familiarity users have with the component modules used in all aspects of a ROCI deployment. By making the same framework pervasive throughout the development pipeline, users are able to concentrate their efforts on improving core techniques because the code only needs to be written once. Once the code has been written, users setting up robot behaviors work from the same toolbox as those formulating queries at run time and throughout post-processing.

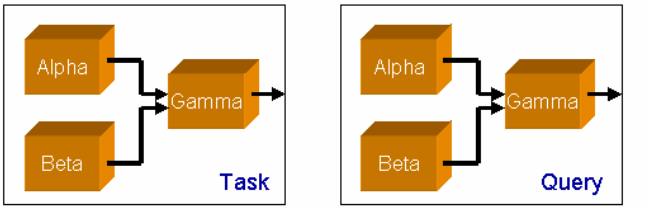

Figure 3. A ROCI query is, in many ways, very similar to a real-time task. In many cases, the inputs of the task come from live sensors, while the query gets data from logs. This distinction is transparent to the component modules.

The notion of query stages combined with the strong type system underlying ROCI module inputs and outputs immediately opens the door for a multitude of queries that make use of functionality already used by robot behaviors. For example, a robust localization routine may be run on all robots as they move around the environment. This routine must update relatively quickly to allow the robot to navigate in real-time, thus necessitating that it only consider readily available data. However, a user or autonomous agent may require an alternate estimation of a robot's location at a particular time in the past, perhaps utilizing newly acquired data. This can be achieved by designing a query wherein a localization routine, possibly another instance of the original routine, is connected to not only locally collected data, but also to any number of data processing routines, also specified by the query body, running on any number of other nodes. This localization may take a relatively long time to execute, and may not be suitable for real time control, but it is available to any programmed behavior or human operator that requests it. This query, while complex, automatically benefits from the shared toolbox provided by the consistent design framework. Processing modules that already exist on data hosts need not be transmitted, while others are downloaded from peers on an as-needed basis. The query itself is analogous to a behavior task: it specifies processing modules and how they connect. The ROCI kernel handles the work of ensuring that modules exist on the nodes that need them, and that those modules are properly connected.

By applying distributed database methods and techniques, the architecture presented here frees designers from having to create a static, all-encompassing communications scheme capable of satisfying a set of pre-specified query types. Instead, individual developers are able to utilize all sensor network resources in a modular, dynamic fashion through the use of active distributed database queries.

3. APPLICATIONS

ROCI technology is being used throughout the GRASP Lab to power a variety of robotics projects. The structure supported by ROCI facilitates the design of complex single-platform systems, high-performance real-time behaviors, and relatively simple static sensors. Projects such as the Smart Wheelchair utilize ROCI to organize and make sense of the data collected by dozens of sensors on a single mobile platform. Teams of small truck-like robots (Clodbusters) use ROCI for everything from collaborative error minimization to vision-based obstacle avoidance. Even a fixed camera becomes far more useful when plugged into a computer running ROCI. ROCI immediately provides logging capabilities as well as the ability to expose the camera's data stream to the network. Teams of ROCI-powered vehicles made up of Clodbusters, fixed wing UAVs, and an autonomous blimp have been successfully fielded in exploration and navigation experiments under adverse network conditions as part of the DARPA-funded MARS2020 program.

Current database-related work involves visualization and exploitation of data generated by a heterogeneous team of ground and air robots equipped with cameras, GPS receivers, IMU readers, altimeters and other sensors. For visualization purposes, this data can be fused in an on-demand fashion through visualization modules a human operator can interact with. In this way, one can quickly bring up images taken by a UAV flying over a particular location by joining a GPS log with an image log over a time index. Of note is what data is sent over the network to meet a particular demand. To minimize network usage, one might use a map location selected by the user to index into a GPS log to see when the robot was at the desired location, if it ever was. The resultant time indices can be used to index into the image database, thus avoiding the need to transfer unnecessary images.

An alternate formulation of this scenario that still maintains network efficiency, while improving usability, is to obtain the time indices of all images taken within some timeframe. These indices can be used to index into the GPS log to present the user with a map marked

up with the locations where pictures were taken. The user can select one of these locations, thus providing the database system with a time index to use in obtaining a particular image. This solution exploits the fact that both time indices and GPS data are far more compact than image data. The goal is to transmit as narrow a subset of the largest data log, in this case the image log, as possible. This setup is what is used at the GRASP Lab to intuitively scan data collected during a team operation.

A behavior-oriented application of the logging functionality can be found in a mobile target acquisition behavior. In this scenario, periodically placed overhead camera nodes log their image data which is made accessible to mobile robots when a network route to the camera node exists. Given a piece of code for visually identifying a target, a mobile robot can move to within routed radio range of overhead camera nodes and inject the target identification code as part of an image log query. The results of this query can simply be the time indices when the target was visible to the overhead camera. This information can be used to improve the efficacy of visual target searches – an extremely data-intensive process -- while minimizing the burden placed on the network. Under lab conditions, a two-node network, using a technology based on 802.11b ad-hoc networks, may be expected to manage 300KB/sec data transfer rates. This would mean that a single, uncompressed 1024x768 color image (2.25MB) would take over 7 seconds to transfer. While compression can greatly help, any resultant artifacts could cripple the effectiveness of a given processing algorithm. Regardless, a factor of 10 gained in compression is more than lost when faced with an image log of thousands of images. Compare this to the 20-50KB size of a typical ROCI module DLL, and it is clear that transferring the code rather than the data often presents considerable advantages.

4. EVALUATION

The primary benefit of ROCI is the development process it suggests. Developing high level applications from reusable, modular components is a well-understood concept, but one whose acceptance has faced real difficulties as popular programming technologies have not kept up with the requirements of modern design techniques. ROCI represents an attempt to push the field forward by taking full advantage of powerful hardware as well as relatively modern programming techniques such as object-oriented programming and strong type systems. By building consistent support for the type system into our high level framework we have successfully allowed loosely structured development teams to collaborate on large-scale projects with more reliable results than is usual. The task-module-pin design structure encourages engineers without strong computer science backgrounds to contribute to larger projects without having to worry about their lack of understanding of the underlying system. Most developers concentrate on the specifics of what their module does, not how it fits into a larger system, or how any of the internal mechanisms – such as scheduling, communications, or user interface – work.

4.1 *Related Work*

Similar systems exist for other application scenarios. TinyOS is an open-source effort to provide OS-level support for sensor platforms with extremely limited hardware. In fact, the fundamental design concepts of TinyOS and ROCI have much in common, primarily the encouraging of modular software design [1]. However, TinyOS specifically targets limited hardware platforms, which imposes limits on what can be attempted with it. We have chosen to target much more capable hardware – we use consumer-level laptop computers on many of our robots – and we are therefore able to distance ourselves from many of the difficulties faced by the mote programmer.

Distributed computing infrastructures that target more powerful hardware can also be found. Several Grid computing efforts are making large strides towards harnessing the computational power of thousands of computers over the Internet [3]. These efforts tend to be of a much more general slant than what we have undertaken. We have found that by focusing on the needs of our developers, we are better able to define constraints on the development process that significantly improve reliability. While the event-based pin communications infrastructure ROCI employs works well for sensor platforms exchanging data, it is not necessarily optimal for all computing needs. Further, we do not provide any tools for automating the distribution of a single computation over a very large network.

Sun's Jini system for Java is an architecture that attempts to bridge the gap between embedded systems and services running on general purpose computers [2]. While this system boasts many of the same benefits as ROCI, we feel that it requires somewhat more effort on the part of the developer to make use of. The simplest ROCI deployments involve minimal usage of ROCI API calls. There is a template module that an author fills out, and then each pin connection in a task is specified in one line of XML. This type of deployment is an example of how the ROCI kernel is designed to handle most common usage patterns with minimal developer action.

4.2 *Final Words*

The ROCI system is evidence that a strong type system paired with solid software design fundamentals can yield substantial improvements in software reliability, reuse, and ease of use. While still primarily used in robotics efforts, projects that seek to stretch ROCI design methods in new directions, such as limited hardware devices and schedule optimization, are now underway. By defining the ROCI kernel itself in a modular fashion with well-defined interfaces, we are able to extend the offered functionality, usually without breaking backwards compatibility. This extensibility, both in terms of novel task-level applications and kernel extensions, is a validation of the design methods presented above.

5. REFERENCES

[1] Philip Levis, Sam Madden, David Gay, Joe Polastre, Robert Szewczyk, Alec Woo, Eric Brewer and David Culler, "The Emergence of Networking Abstractions and Techniques in TinyOS," Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004).

[2] Jim Waldo, "The Jini Architecture for Network-centric Computing," *Communications of the ACM*, pp. 76-82, July 1999.

[3] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri, "A Component Based Services Architecture for Building Distributed Applications," In Proc. 9th IEEE International Symposium on High Performance Distributed Computing, Pittsburgh, PA, Aug. 2000.

[4] R. Alur, A. Das, J. Esposito, R. Fierro, G. Grudic, Y. Hur, V. Kumar, I. Lee, J. Ostrowski, G. Pappas, B. Southall, J. Spletzer, and C. Taylor, "A framework and architecture for multirobot coordination," In D. Rus and S. Singh, editors, *Experimental Robotics VII, LNCIS 271*. Springer Verlag, 2001.

[5] Sarah Bergbreiter and K.S.J. Pister, "Cotsbots: An off-the-shelf platform for distributed robotics," In *IROS*, pp. 1632, October 2003.

[6] Luiz Chaimowicz, Anthony Cowley, Vito Sabella, and Camillo J. Taylor, "Roci: A distributed framework for multi-robot perception and control," In *IROS*, pp. 266, 2003.

[7] A. Das, J. Spletzer, V. Kumar, and C. J. Taylor, "Ad hoc networks for localization and control," Proceedings of the IEEE Conference on Decision and Control, 2002.

[8] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," *International Journal of Supercomputer Applications*, 15(3), 2001.

[9] A. Cowley, H. Hsu and C.J. Taylor, "Distributed Sensor Databases for Multi-Robot Teams," Proceedings of the 2004 IEEE Conference on Robotics and Automation (ICRA), April 2004.

[10] Joseph M. Hellerstein, Wei Hong, Samuel Madden, and Kyle Stanek, "Beyond average: Towards sophisticated sensing with queries," In *Information Processing in Sensor Networks*, March 2003.

[11] D. MacKenzie, R. Arkin, and J. Cameron, "Multiagent mission specification and execution," *Autonomous Robots*, 4(1): pp. 29-52, 1997.

[12] Samuel R. Madden, Michael J. Franklin, Joseph Hellerstein, and Wei Hong, "The design of an acquisitional query processor for sensor networks," In *SIGMOD*, June 2003.

[13] D. Martin, A. Cheyer, and D. Moran, "The open agent architecture: a framework for building distributed software systems," *Applied Artificial Intelligence*, 13(1/2): pp. 91-128, 1999.